Simplifying TugGraph using Zipping Algorithms

S. Golodetz[a,*], A. Arnab[b,1], I.D. Voiculescu[c], S.A. Cameron[c]

[a] *Oxford Research Group, FiveAI Ltd., Oxford, United Kingdom*
[b] *Department of Engineering Science, University of Oxford, United Kingdom*
[c] *Department of Computer Science, University of Oxford, United Kingdom*

Abstract

Graphs are an invaluable modelling tool in many domains, but visualising large graphs in their entirety can be difficult. Hierarchical graph visualisation – recursively clustering a graph's nodes to view it at a higher level of abstraction – has thus become popular. However, this can hide important information that a user needs to understand a graph's topology, e.g. nodes' neighbourhoods. TugGraph addressed this by 'separating out' a given node's neighbours from their hierarchy ancestors to visualise them independently. Its original implementation was straightforward, but copied parts of the hierarchy, making it slow and memory-hungry. An optimised later version, which we refer to as *FastTug*, avoided this, but at a cost in clarity. Optimising TugGraph without sacrificing clarity is difficult because of the need to keep every hierarchy node connected, a common challenge for graph hierarchy editing algorithms. Recently, this problem has been addressed by 'zipping' algorithms, multi-level split/merge algorithms that preserve hierarchy node connectedness and can be built upon for higher-level editing. In this paper, we generalise the original unzipping algorithms to implement *SimpleTug*, a simple, modular version of TugGraph that is easy to understand and implement, and even faster and more memory-efficient than *FastTug*. We formally prove its equivalence to *FastTug*, and show how both can be parallelised. Using our *millipede* hierarchical image segmentation system, we show experimentally that both the serial and parallel versions of *SimpleTug* are

*Corresponding author
Email addresses: stuart@five.ai (S. Golodetz), irina@cs.ox.ac.uk (I.D. Voiculescu)
[1] A. Arnab is now with Google Research.

around 25% faster than their *FastTug* counterparts, whilst using considerably less memory. Finally, we discuss the interesting theoretical connections between TugGraph and zipping, and suggest ideas for further work.
Keywords: graph visualisation, TugGraph, cluster hierarchy, zipping algorithms.

---

1. Introduction

Graphs are a fundamental mathematical modelling tool because of their ability to represent both the entities relevant to a problem and the relationships between them. Having modelled a problem's entities using a graph, it is common to want to visualise the results to gain greater insight into the underlying problem [1, 2], but many modelling tasks – e.g. mapping the connections between all of the servers on the Internet or between all of the people on a social network, analysing the architecture of a large software system, or visualising a deep neural network [3] – can involve extremely large graphs that can be difficult, or even impossible, to visualise in their entirety.

The obvious solution to this problem is to condense the large amount of information available in the graph as a whole into a smaller amount of data that can be presented to the user, a task that can be approached in various ways. One simple approach might be to select a subgraph of the overall graph consisting of all nodes that satisfy a particular property, together with their associated edges. A downside of this is that the subgraph contains no information whatsoever about the remaining nodes of the overall graph.

An alternative approach that avoids this problem is to visualise the graph *hierarchically*, recursively clustering its nodes into *aggregate* nodes, each of which summarises the information contained in the nodes below it in the hierarchy.[2] Hierarchical graph visualisation has long been popular because of its ability to summarise the entirety of the underlying graph, and significant research efforts have been devoted to developing appropriate user interfaces that can maximise its effectiveness. Common ways of visualising graph hierarchies include node-

link diagrams [4], adjacency matrices [5], treemaps [6] and radial techniques [7], among many others [8]. Hybrid approaches, which aim to leverage the advantages of several of these techniques at once, have also proved popular. For example, [9] combines node-link diagrams with treemaps, whilst [10] and [11] combine them with matrices. A hybrid method is commonly adopted when using a single technique would cause some information to be poorly visualised or even missed. For example, a recent approach by Angori et al. [12] uses chord diagrams to visualise locally dense subgraphs of a globally sparse network, but a node-link diagram to represent the overall graph.

Beyond pure visualisation, much attention has also been devoted to thinking about the interactive tasks a typical user might need to perform when exploring a graph hierarchy, e.g. aggregate selection [17], zooming, panning, or 'drilling down' to explore the contents of an aggregate node [8]. One common task that users might want to perform [18] is to explore the local neighbourhood of a particular node, e.g. a network administrator might want to find the individual servers that are directly connected to his/her own. These kinds of details can be hidden when information is aggregated: for example, it might be easy to see that the server is connected to ones in another country, but impossible to know their exact locations without drilling down further. This is the problem addressed by the popular TugGraph algorithm [18], which works by 'separating out' the neighbours of a given node from their ancestors in the hierarchy to allow them to be visualised independently. The original implementation of TugGraph, whilst easy to understand, was comparatively slow and memory-hungry because of the way in which it unnecessarily copied parts of the hierarchy so that they could be restored later. The same authors [19] later presented an optimised

[2]Many techniques exist for actually constructing graph hierarchies. Typically, approaches are based on either recursive clustering or splitting: e.g. see [13] for some references. Two interesting recent directions in this area have been in learning hierarchical graph representations using graph neural networks (GNNs) [14], and in collaboratively segmenting multimodal images [15, 16]. Works such as these complement the work described in this paper, in the sense that as graph hierarchies are put to an ever-broader range of uses, the need for techniques to visualise (and edit) them efficiently becomes increasingly important.

implementation (which we will refer to as *FastTug*) that avoided this copying, but at a significant cost in algorithmic clarity.

The key challenge when optimising TugGraph is to ensure that an invariant – that every node in the resulting hierarchy is connected – remains satisfied at the end of the operation. *FastTug* achieves this, but only by initially breaking this invariant and then restoring it over the course of the entire operation, which has at least two consequences. Firstly, the nodes whose connectivity is initially broken by *FastTug* must be 'fixed up' by the end of the operation, and storing which nodes to fix up can take significant memory. Secondly, the *FastTug* algorithm can only be understood monolithically: if the hierarchy is examined at any point during the operation, the invariant will not be satisfied. Since decomposing large algorithms into smaller pieces is one of the key ways in which humans seek to understand their functionality [20], this in practice makes *FastTug* significantly less clear as an algorithm than the original TugGraph.[3]

The root cause of these problems is that *FastTug*, in common with many graph hierarchy editing algorithms, is effectively attempting to do two things at once: (i) effect the desired changes to the graph hierarchy, and (ii) preserve the connectedness of the hierarchy's nodes (i.e. the invariant). This represents a poor separation of concerns, and in this case yields an algorithm that is unnecessarily hard to understand. Recent work [13], however, has shown how to solve such problems by introducing 'zipping' algorithms, multi-level split and merge algorithms for hierarchies that preserve the connectedness of the hierarchy nodes and serve as modular building blocks for higher-level editing algorithms.

In this paper, we show how to reformulate TugGraph in terms of these algorithms, yielding an approach that is significantly faster than *FastTug* and uses considerably less memory, whilst retaining the algorithmic clarity of the origi-

[3]Though beyond this paper's scope, a further issue is that the monolithic tugs of *FastTug* cannot be interleaved, since each tug can only start at a point at which the hierarchy invariant is satisfied: if multiple users want to tug on the same hierarchy concurrently (e.g. in a cloud visualisation context), the tugs must either all be started at the same time, or be serialised, which would impede performance. In this paper, we show how tugs that start together can be parallelised, but leave interleaving, which is significantly more involved, for future work.

nal TugGraph approach. Our approach, which we call *SimpleTug*, is modular, easy to implement, and can be expressed in closed form, shedding insight on how *FastTug* itself works, and on the interesting theoretical connections between TugGraph and zipping. We formally prove its equivalence to *FastTug*, and show how both can be parallelised to achieve greater speed. Using our *millipede* hierarchical segmentation system [13], we show experimentally that average tugging operations using both serial and parallel versions of *SimpleTug* are around 25% faster than with *FastTug*, and use less memory.

The rest of the paper is organised as follows: in §2, we review the definition of graph hierarchies from [13], and related work on graph hierarchy exploration, before discussing the original TugGraph algorithm, *FastTug*, and the original unzipping algorithm on which we will base our approach to simplify TugGraph; in §3, we show how unzipping can be generalised to support non-horizontal cuts; in §4, we present our *SimpleTug* method, analyse its complexity and prove its equivalence to *FastTug*; in §5, we describe how to parallelise both *SimpleTug* and *FastTug*; in §6, we demonstrate the practical effectiveness of *SimpleTug* via quantitative experiments; in §7, we discuss the interesting theoretical connections between TugGraph and zipping; and in §8, we conclude.

2. Background

The TugGraph algorithm [18] was originally designed as a visualisation technique for interactively exploring the local neighbourhood of nodes in a graph hierarchy. Visualising graph data hierarchically per se is a huge topic that is beyond the scope of the present paper, although there are relevant surveys, e.g. on hierarchical aggregation for information visualisation [8], visual analysis of large graphs [21], scalable graph exploration and visualisation [22], and visualising group structures in graphs [23]. In this section, we will restrict ourselves to briefly reviewing the definition of graph hierarchies from [13] and surveying existing graph hierarchy exploration techniques, before discussing TugGraph itself and the zipping algorithms that we use to simplify it in more detail.

Structure of graph hierarchy $\mathcal{H}$	
Base graph	$G_{\mathcal{H}} = (V_{\mathcal{H}}, E_{\mathcal{H}})$
All nodes in hierarchy	$V^*_{\mathcal{H}} (\supseteq V_{\mathcal{H}})$
All nodes at depth d	$V^d_{\mathcal{H}} (\subseteq V^*_{\mathcal{H}})$
Depth / region operations on hierarchy nodes	
Combined region of one or more nodes	$\mathcal{R}(n)$ / $\mathcal{R}(N)$
Common depth (if any) of one or more nodes	$\mathcal{D}(n)$ / $\mathcal{D}(N)$
Maximum depth of any node in set N	$\mathcal{D}_{\max}(N)$
Minimum depth of any node in set N	$\mathcal{D}_{\min}(N)$
Ancestor / descendant operations on hierarchy nodes	
Single common ancestor at depth d, if any	$\psi^d_{\mathcal{H}}(n)$ / $\psi^d_{\mathcal{H}}(N)$
All ancestors / descendants at depth d	$\Psi^d_{\mathcal{H}}(n)$ / $\Psi^d_{\mathcal{H}}(N)$
All ancestors (regardless of depth)	$\Psi^+_{\mathcal{H}}(n)$ / $\Psi^+_{\mathcal{H}}(N)$
All descendants (regardless of depth)	$\Psi^-_{\mathcal{H}}(n)$ / $\Psi^-_{\mathcal{H}}(N)$
All descendants in set N	$\Psi^-_{\mathcal{H}\|N}(n)$
Node set filtering operations	
Nodes in set N at depth d	$N_{=d}$
Nodes in set N at depth $> d$	$N_{>d}$

Table 1: The notation used throughout this paper. Note that most operations are overloaded to allow them to be applied to both individual nodes and node sets. For example, $\Psi^+_{\mathcal{H}}(n)$ returns all ancestors of node n in hierarchy $\mathcal{H}$, whilst $\Psi^+_{\mathcal{H}}(N)$ returns all ancestors of any node in node set N.

2.1. Graph Hierarchies

As defined in detail in [13], a graph hierarchy $\mathcal{H}$ is a data structure that results from recursively clustering the nodes of an undirected graph $G_{\mathcal{H}} = (V_{\mathcal{H}}, E_{\mathcal{H}})$, in which $V_{\mathcal{H}}$ and $E_{\mathcal{H}}$ represent the nodes and edges of $G_{\mathcal{H}}$, respectively. We call $G_{\mathcal{H}}$ itself the *base graph* of $\mathcal{H}$, since it is the graph on which $\mathcal{H}$ is based. Clustering the nodes in $V_{\mathcal{H}}$ hierarchically yields the set $V^*_{\mathcal{H}} \supseteq V_{\mathcal{H}}$ of all nodes in hierarchy $\mathcal{H}$, where each node in $V^*_{\mathcal{H}}$ represents the aggregation of its descendants in $V_{\mathcal{H}}$. Structurally, we can write each node $n \in V^*_{\mathcal{H}}$ as a pair $(\mathcal{D}(n), \mathcal{R}(n))$, where $\mathcal{D}(n)$ denotes the *depth* of n in $\mathcal{H}$ (defined as 0 for the root node, and one plus the depth of the node's parent otherwise), and $\mathcal{R}(n)$ denotes the *region* of n (defined as the set of n's descendants in $V_{\mathcal{H}}$). We denote the set of all nodes at depth d as $V^d_{\mathcal{H}}$.

As mentioned in §1, a key invariant of the graph hierarchies we consider for TugGraph [18] is that each node in a hierarchy $\mathcal{H}$ must be connected, or more formally that for each node $n \in V^*_{\mathcal{H}}$, the subgraph of $G_{\mathcal{H}}$ induced by $\mathcal{R}(n)$ must be connected. As noted, *FastTug* [19] temporarily breaks this property

before restoring it at the end of the operation (see §2.4), whereas our approach preserves it throughout the process (see §4).

To help with later definitions, we follow [13] in extending the depth/region concepts to a set of nodes N. We define $\mathcal{R}(N) = \bigcup \{R(n) : n \in N\}$ to be the union of the regions of the nodes in N, and $\mathcal{D}_{\max}(N)$ and $\mathcal{D}_{\min}(N)$ (respectively) to be the maximum and minimum depths of any node in N. Furthermore, if the nodes in N have a common depth, i.e. $\mathcal{D}_{\max}(N) = \mathcal{D}_{\min}(N)$, we denote this as $\mathcal{D}(N)$. Finally, we also follow [13] in defining a number of functions that allow us to denote the ancestors or descendants of a node or a set of nodes, either in general or at a specified depth d (see Table 1).

One concept that was not discussed in [13] is that of a *cut* through a hierarchy: for our purposes here, we define a cut $\mathcal{C}$ through a hierarchy $\mathcal{H}$ to be a set of non-overlapping nodes in $V^*_{\mathcal{H}}$, not necessarily all at the same depth, such that $\mathcal{R}(\mathcal{C}) = \mathcal{R}(V_{\mathcal{H}})$.

2.2. Graph Hierarchy Exploration

Existing methods for exploring graph hierarchies can be broadly divided into two types:

Passive (view-based) methods change how the hierarchy is being viewed without modifying it. For example, Klava et al. [24]'s split/merge operations modify the non-horizontal cut used to view a hierarchy to change which regions are displayed. Abello et al. [25] present a system called ASK-GraphView, which is based on a graph hierarchy model and supports both node-link and hierarchical visualisations. They focus on *interactive* visual data exploration [26], supporting zooming, panning, expanding/collapsing nodes, filtering and colouring. Tominski et al. [27] present a successor to ASK-GraphView called CGV, which is similarly based on a graph hierarchy model and adds a 'magic eye' visualisation, together with fisheye magnification and the ability to locate a node in multiple synchronised views. They use lenses to remove node/edge clutter and gather adjacent nodes, and introduce an interesting data-space nav-

igation technique called 'edge-based travelling'. Gladisch et al. [28] recommend where to navigate next in a hierarchy to satisfy a particular goal. They select candidate nodes around the currently viewed nodes, rank them, and then direct the user towards the highest-ranked nodes using visual cues. Zhang et al. [29] make it easier to maintain context when moving between hierarchy scales by visualising a sliding window of scales in the same space, coupling a nested map with a tree view to clarify what is being visualised. Inspired by similar considerations, AlTarawneh et al. [30] present a way of expanding nodes that preserves the overall layout of the unexpanded nodes to avoid disrupting the user's mental map.

Active (model-based) methods, by contrast, physically modify the hierarchy to make relevant information easier to see. For example, various works [31, 32, 33] group/ungroup nodes in a hierarchy to better reflect the relationships between them. Other common operations include merging/splitting nodes [34], changing a node's parent [35], and morphological flooding [36] (this is used to reduce oversegmentation in a graph hierarchy for an image, to make the high-level structure of the image easier to see). Archambault et al. [4] describe an active method called Reform-Below-Cut, which deletes part of the hierarchy below the cut being viewed and recreates it in such a way as to group together lower-level nodes that share properties. Maire et al. [37] present an interactive hierarchy editing system that allows regions to be dragged and dropped to new parent nodes so as to model object-part relationships in a way that better matches the user's understanding. In very recent work, Richer et al. [38] present a prototype of a distributed hierarchy exploration and editing system based on parallel coordinates that can scale to billion-item datasets.

2.3. TugGraph

TugGraph [18] itself is an active visualisation algorithm that can be used to modify a graph hierarchy so as to 'separate out' the parts of the base graph that are adjacent to a selected node on the non-horizontal cut (see §3) that is

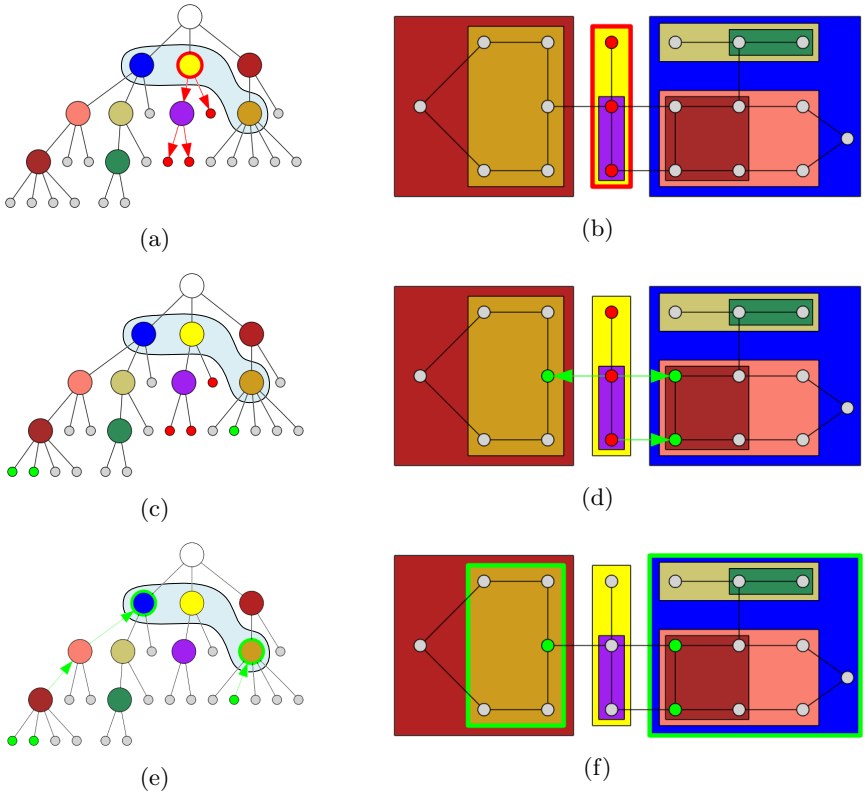

Figure 1: An example of steps 1–3 of the TugGraph algorithm [18] on a simple graph hierarchy. We render the hierarchy in two different ways: as a tree in (a,c,e), and as a clustering of the underlying graph in (b,d,f). The pale blue loop in (a,c,e) denotes the cut that is being used to view the hierarchy (this comes from the user interface being used). The algorithm takes as input this cut and a user-selected node (the bright yellow node surrounded by a red border in (a,b)). First, we traverse down the hierarchy from this node to find the leaves in its subtree (coloured in red); (c,d) then, we find the leaves adjacent to those leaves in the underlying graph (the 'adjacent leaves', coloured in green); (e,f) finally, we traverse back up the hierarchy from the adjacent leaves to find the nodes on the cut that are adjacent to the original node (the 'adjacent cut nodes', circled in green).

currently being viewed. The analogy is of 'tugging' on the selected node in order to pull adjacent nodes that are currently hidden into view, thereby allowing a user to explore the topological structure around the node. The algorithm takes a single node and a cut as input, and works as follows:

1. First, it traverses down the hierarchy from the selected node to find the leaves it subsumes.

2. Then, it finds the leaves adjacent to those leaves in the base graph (the 'adjacent leaves').

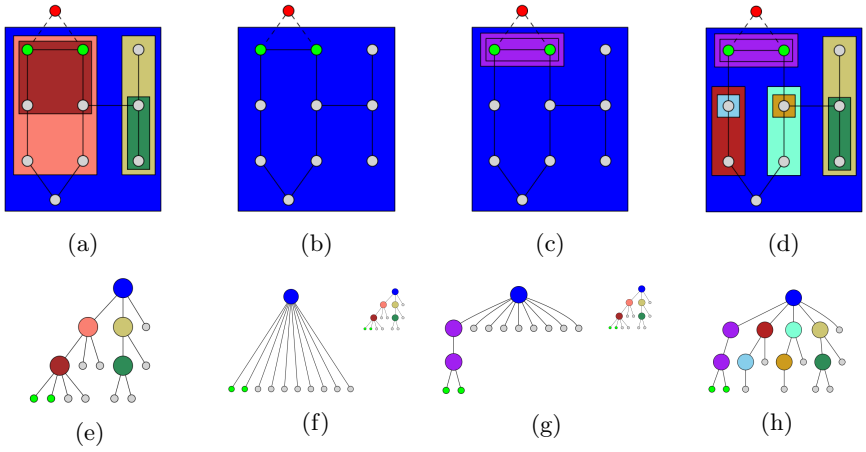

Figure 2: The effects of steps 4–5 of the original TugGraph [18] on a single adjacent cut node: (a,e) we 'tug' on the node at the top of the diagram (red), with the intention of 'separating out' the adjacent leaves (green) within the adjacent cut node (blue); (b,f) the intermediate hierarchy between the adjacent cut node and the leaves it subsumes is copied (see the smaller tree) and then deleted; (c,g) the adjacent leaves are grouped together based on their connectivity in the graph; (d,h) the intermediate hierarchy between the adjacent cut node and all other leaves is recreated based on the previously-made copy (this involves recursively applying the Reform-Below-Cut operation from GrouseFlocks [4] bottom up [19]).

3. Next, it traverses up the hierarchy from the adjacent leaves to find the nodes on the cut that are adjacent to the original node (the 'adjacent cut nodes').

4. For each adjacent cut node, it makes a copy of and then deletes the intermediate hierarchy between it and the leaves it subsumes, and then groups the adjacent leaves it now directly contains together based on their spatial connectivity in the base graph.

5. Finally, for each adjacent cut node, it recreates the intermediate hierarchy between it and all other leaves it contains based on the previously-made copy.

An example showing steps 1–3 of this process can be seen in Figure 1. In Figure 2, steps 4–5 are then illustrated on one of the adjacent cut nodes from this example.[4] The overall effect of the operation is to create new nodes, just below the cut, that together contain all the leaves adjacent to the originally-selected node, and the visualisation can subsequently be adjusted to show them.

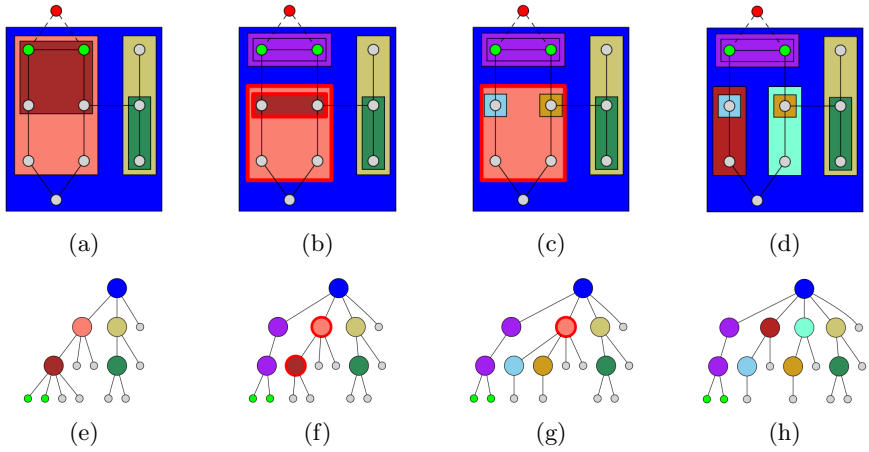

Figure 3: The effects of *FastTug* on the same example shown in Figure 2. In (b,f), the adjacent leaves (green) are directly moved into a node just below the cut, disconnecting their former ancestors (highlighted in red) and thereby temporarily breaking the invariant that every hierarchy node is connected. In (c,g) and (d,h), the hierarchy is then fixed by splitting first the former parent (c,g) and then the former grandparent (d,h) of the adjacent leaves.

Conceptually, this is relatively straightforward, but implementing the tugging process in this way is quite inefficient because it has to recreate the intermediate hierarchy for all leaves within an adjacent cut node, even if they are not relevant to the node that has been selected (for example, the right-hand part of the hierarchy in Figure 2 is unchanged by the operation, but this implementation deletes it and recreates it from scratch).

2.4. Optimised TugGraph (FastTug)

To improve the efficiency of the original TugGraph, the same authors later published an optimised version of their approach [19] that avoids copying and later recreating the hierarchy in steps 4–5 of the algorithm. This optimised version, which we refer to here as *FastTug*, replaces these steps with the following alternative approach (see also Figure 3):

[4]Attentive readers will note that some of the nodes created in our examples have only a single child, meaning that the resulting hierarchy ends up containing multiple nodes that represent the same portion of the base graph. There is no fundamental difference between such a hierarchy and one in which all node chains involving single children have been compressed, but preserving the chains avoids the need for any nodes to change depth, which allows *FastTug* and *SimpleTug* to be implemented in a layer-based segmentation system based on complete hierarchies (such as the one described in §6). In a non-layer-based system, it might be desirable to compress these node chains, either in the representation itself (after a tug), or in the UI.

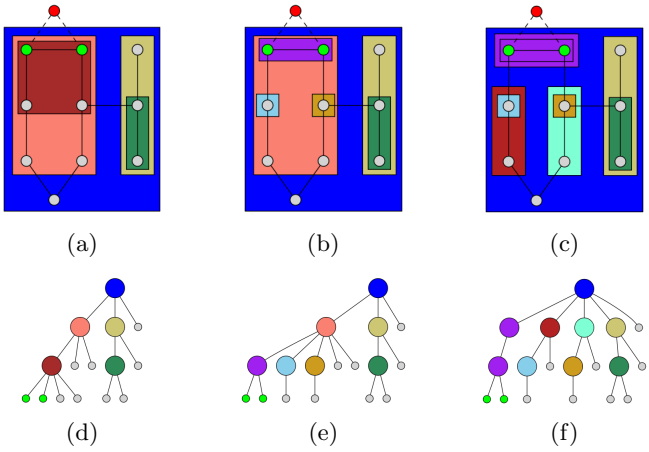

Figure 4: The effects of *SimpleTug* (see §4) on the same example shown in Figures 2 and 3. We reformulate the tugging operation as a multi-node unzip [13] that iteratively separates out the adjacent leaves (green) from their ancestors in the hierarchy. In (b,e), the brown node is split around them, separating them out from their original parent. Then, in (c,f), the pink node is split around their new parent (purple), separating them out from their original grandparent. *SimpleTug* achieves the same result as TugGraph and *FastTug*, but avoids copying the hierarchy or breaking the hierarchy invariant during the operation.

4) After step 3, it first moves the adjacent leaves directly into nodes just below the cut. As shown in Figure 3(b), this has the potential to disconnect all of the ancestors of these leaves below the cut, temporarily breaking the invariant that every node in the hierarchy is connected.

5) Having done this, it then works up the parts of the hierarchy below the adjacent cut nodes, splitting nodes and updating adjacency information as necessary to restore the invariant (see Figures 3(c) and (d)).

FastTug markedly improves the efficiency of TugGraph [19], but as an algorithm, it is harder to understand and more difficult to implement. As mentioned in §1, this is ultimately caused by the way in which *FastTug* can potentially break the hierarchy invariant in step 4, only restoring it again at the end of the algorithm. This prevents us from decomposing the algorithm into smaller, invariant-preserving pieces that are easier to understand and implement in isolation. In §4, we will show how these problems can be overcome by reformulating TugGraph in terms of the zipping algorithms described in [13]. At a high level, this will involve changing the way in which the adjacent leaves are separated out

Ancestor Splitting (adapted from [13])

$$
\begin{aligned}
\text{tugged}(d, N) &= \{(d, R') : R' \in \text{ccs}(\mathcal{R}(N))\} \\
\text{remnantRegion}(n, N) &= \mathcal{R}(n) \setminus \mathcal{R}(N) \\
\text{remnants}(n, N) &= \{(\mathcal{D}(n), R') : R' \in \text{ccs}(\text{remnantRegion}(n, N))\} \\
\text{splitAnc}_{\mathcal{H}}(d, N) &= \text{tugged}(d, N) \cup \text{remnants}(\psi^d_{\mathcal{H}}(N), N)
\end{aligned}
$$

Table 2: An adaptation of the ancestor-splitting operation on which unzipping relies (see §2.5). Unlike the original [13], this version is easier to compare with the corresponding 'ancestor-ripping' operation in *FastTug* (see Table 7). Informally, `splitAnc` splits n, the common ancestor of the nodes in N at depth d, around those nodes, yielding nodes corresponding to the connected components ('ccs') of $\mathcal{R}(N)$, and nodes corresponding to the connected components of $\mathcal{R}(n) \setminus \mathcal{R}(N)$.

from their ancestors so as to preserve the invariant after each individual split (see Figure 4). This initial intuition will be pinned down more formally once we have described how our zipping algorithms work in the following sections.

2.5. Zipping Algorithms for Graph Hierarchy Editing

In contrast to TugGraph, the zipping algorithms in [13] were not designed for hierarchy visualisation, but rather as general-purpose building blocks on which to build algorithms for hierarchy editing. Two primary operations were described: *unzipping*, which takes as input one or more nodes and separates them out from some or all of their ancestors in the hierarchy, and *zipping*, which merges chains of nodes a layer at a time, in order of increasing depth. These operations can in some sense be seen as multi-layer equivalents of node splitting and (sibling) node merging. However, unlike a general multi-layer split, which would have significant input requirements (the user would have to specify the pieces into which each node in each layer would need to be split), unzipping works by splitting the ancestors of one or more nodes around those nodes. This reduces the input needed from the user to just the set of nodes to unzip, and eliminates the need for the user to understand the structure of the hierarchy in any detail. In this section, we briefly summarise how unzipping works, before showing how it can be generalised and used to simplify TugGraph in Section 4. Further details about the zipping algorithms themselves can be found in [13].

The ancestor-splitting operation on which unzipping relies, called `splitAnc`,

Figure 5: An illustration of the ancestor splitting operation on which unzipping relies: (a) the ancestor node $n = \psi^d_{\mathcal{H}}(N)$ being split (at depth d); (b) the descendants of n at some depth $d' > d$; (c) the descendants N at depth d' around which to split the ancestor (highlighted in red) and n's remaining descendants at depth d' (highlighted in blue); (d) the depth d nodes resulting from the split. Note that this figure is an adapted version of our Figure 4 from [13], modified to reflect the formulation used in this paper.

can be defined as shown in Table 2. Unlike the definition of `splitAnc` in [13], this version has the advantage that it can be clearly contrasted with the corresponding 'ancestor-ripping' operation in *FastTug* [19] that is used when moving adjacent leaves directly into nodes just below the cut (see §2.4 and Table 7).

The version we give here takes two parameters – a set of nodes N and a depth d at which the nodes in N have a common ancestor n – and splits this common ancestor around the nodes in N. To do this, it first computes the connected components of $\mathcal{R}(N)$ and defines a depth d node for each such connected component (we refer to these nodes collectively as $\mathit{tugged}(d, N)$, since they can be seen as the nodes 'tugged' out of n by the split). It then computes the connected components of what is left of n (namely $\mathcal{R}(n) \setminus \mathcal{R}(N)$), and again defines a depth d node for each such connected component (we refer to these latter nodes collectively as $\mathit{remnants}(n, N)$). Finally, it yields the union of these two sets of connected components as the result of the split. An illustration can be seen in Figure 5.

Unzipping itself, in its original formulation, can then be defined as shown in Table 3. It takes a set of nodes N and a depth $d_{\min}$ to which to unzip them, and works layer-by-layer up the hierarchy from depth $\mathcal{D}_{\max}(N) - 1$ to depth

Classic Multi-Node Unzipping to a Specified Depth (adapted from [13])

Inputs: N, the set of nodes to unzip
$d_{\min}$, the depth to which to unzip them (equivalent to a horizontal cut)

$$\mathrm{unzip}_{N;d_{\min}}(\mathcal{H}) = \left(\mathrm{unzip}_{N;d_{\min}}(V^d_{\mathcal{H}}) : V^d_{\mathcal{H}} \in \mathcal{H}\right)$$

$$\mathrm{unzip}_{N;d_{\min}}(V^d_{\mathcal{H}}) = \begin{cases} (V^d_{\mathcal{H}} \setminus \Psi^d_{\mathcal{H}}(N_{>d})) \cup \\ \quad \bigcup \left\{ \mathrm{splitAnc}_{\mathcal{H}}(d, \Psi^-_{\mathcal{H}|N}(n)) : n \in \Psi^d_{\mathcal{H}}(N_{>d}) \right\} & \text{if } d \in (d_{\min}, \mathcal{D}_{\max}(N)) \\ V^d_{\mathcal{H}} & \text{otherwise} \end{cases}$$

Table 3: The classic multi-node unzipping operation from [13], which allows a set of nodes N to be unzipped up to a single specified depth $d_{\min}$ in a hierarchy (effectively a horizontal cut).

$d_{\min}+1$, replacing every ancestor of the nodes in N below a depth of $d_{\min}$ with the results of splitting it around its descendants in N.[5] At depth d, the unzip operation first removes from $V^d_{\mathcal{H}}$ the ancestors of the nodes in N at that depth, written as $\Psi^d_{\mathcal{H}}(N_{>d})$ (some nodes in N may have a depth $\leq d$, which is why we use $N_{>d}$ here to remove them from consideration). For each removed node n, it then computes its descendants $\Psi^-_{\mathcal{H}|N}(n)$ in N, and re-adds the results of splitting n around these descendants to $V^d_{\mathcal{H}}$. For an illustration of this process, we refer the reader to [13].

3. Generalised Multi-Node Unzipping

One limitation of the formulation of unzipping in Table 3 is that it only allows nodes to be unzipped up to a single specified depth $d_{\min}$ (i.e. a horizontal cut at depth $d_{\min}$). However, it is sometimes important to be able to unzip nodes in different parts of the hierarchy to different depths (e.g. when reformulating TugGraph in §4).

One way of achieving this is to perform a sequence of separate unzips, each to a single depth, but whilst this is practically effective, its natural formulation, as

[5] Note that the $d_{\min}$ used in this formulation is subtly different to the $d_{\min}$ used in the formulation in [13] (specifically, it is equivalent to the original $d_{\min}-1$). This makes no difference to the range of unzip operations that can be expressed, but makes it easier for us to generalise the operation to non-horizontal cuts in §3.

Generalised Multi-Node Unzipping (naïve version)	
Inputs:	$\mathbf{N} = (N_1, ..., N_k)$, the sets of nodes to unzip $\mathbf{d}_{\min} = \left(d^1_{\min}, ..., d^k_{\min}\right)$, the depths to which to unzip them, respectively
Hierarchy Transformation Sequence:	$(\mathcal{H} =)\mathcal{H}_0 \overset{u}{\rightsquigarrow} \mathcal{H}_1 \overset{u}{\rightsquigarrow} \ldots \overset{u}{\rightsquigarrow} \mathcal{H}_k$
Intermediate Hierarchies:	$\forall i \geq 1 \,.\, \mathcal{H}_i = \text{unzip}_{N_i;d^i_{\min}}(\mathcal{H}_{i-1})$
	$\text{unzip}_{\mathbf{N};\mathbf{d}_{\min}}(\mathcal{H}) = \mathcal{H}_k$

Table 4: A naïve attempt to generalise the classic multi-node unzipping algorithm from Table 3 and [13] to allow nodes in different parts of the hierarchy to be unzipped to different depths.

Generalised Multi-Node Unzipping to a Non-Horizontal Cut	
Inputs:	N, the set of nodes to unzip $\mathcal{C}$, the non-horizontal cut to which to unzip them
Auxiliary Definitions:	$\Omega^d_{\mathcal{H}}(N, \mathcal{C}) = \Psi^d_{\mathcal{H}}(N_{>d}) \cap \Psi^-_{\mathcal{H}}(\mathcal{C})$
Main Definition:	$\text{unzip}_{N;\mathcal{C}}(\mathcal{H}) = \left(\text{unzip}_{N;\mathcal{C}}(V^d_{\mathcal{H}}) : V^d_{\mathcal{H}} \in \mathcal{H}\right)$
	$\text{unzip}_{N;\mathcal{C}}(V^d_{\mathcal{H}}) = \begin{cases} (V^d_{\mathcal{H}} \setminus \Omega^d_{\mathcal{H}}(N, \mathcal{C})) \cup \\ \quad \bigcup \left\{ \text{splitAnc}_{\mathcal{H}}(d, \Psi^-_{\mathcal{H}\mid N}(n)) : n \in \Omega^d_{\mathcal{H}}(N, \mathcal{C}) \right\} & \text{if } d \in (\mathcal{D}_{\min}(\mathcal{C}), \mathcal{D}_{\max}(N)) \\ V^d_{\mathcal{H}} & \text{otherwise} \end{cases}$

Table 5: A better way of generalising the multi-node unzipping operation from Table 3 and [13]: we allow a set of nodes N to be unzipped up to a *non-horizontal* cut $\mathcal{C}$, rather than a single specified depth $d_{\min}$ (which is effectively a horizontal cut). To achieve this, we modify the definition to split only ancestors of N that are below $\mathcal{C}$. Note that an unzip to a specified depth $d_{\min}$ can still be achieved using this more general formulation by setting $\mathcal{C}$ to $\mathcal{V}^{d_{\min}}_{\mathcal{H}}$.

shown in Table 4, is then iterative rather than being in closed form: this makes it harder to use for proofs, and makes characterising the overall behaviour of the algorithm more difficult. In practice, a better alternative is to generalise the original unzipping algorithm to allow nodes to be unzipped up to a non-horizontal cut of the hierarchy.

To achieve this, we first modify the formulation in Table 3 so that instead of taking a fixed depth $d_{\min}$ to which to unzip, it takes a non-horizontal cut $\mathcal{C}$ (see Table 5). Since there is no longer a fixed depth at which to stop the unzip, we then define the set of depths that are potentially affected by the unzip to

range from $\mathcal{D}_{\max}(N) - 1$ to $\mathcal{D}_{\min}(\mathcal{C}) + 1$, i.e. we use the shallowest depth of a node in the cut to define the shallowest depth that may be affected by the unzip. Finally, we modify the definition to replace the set of nodes $\Psi^d_{\mathcal{H}}(N_{>d})$ that was potentially being split in each layer d with a new set $\Omega^d_{\mathcal{H}}(N, \mathcal{C})$, which restricts $\Psi^d_{\mathcal{H}}(N_{>d})$ to just those nodes that are descendants of $\mathcal{C}$, i.e. nodes that are below the non-horizontal cut. This has the effect of stopping the unzip when the non-horizontal cut is reached.

From an implementation perspective, this new algorithm is a straightforward modification of the original. It simply needs to check each node before splitting it to see whether it is on the cut, and to then avoid further considering its ancestors if so. However, the key thing that we have gained by doing this (rather than performing a sequence of separate unzips) is an ability to express the unzipping of nodes in different parts of the hierarchy to different depths in closed form. As we will see in §4, this will allow us to produce a closed form definition for our *SimpleTug* approach, significantly simplifying the proof that our method is equivalent to *FastTug* in §4.2.

As a final observation, we note that the new unzipping algorithm can still be used to unzip nodes to a specified depth $d_{\min}$ if desired: this can be achieved straightforwardly by setting $\mathcal{C}$ to $V^{d_{\min}}_{\mathcal{H}}$. In that sense, it is a true generalisation of the original algorithm. In practice, however, it is not a good idea to specify a horizontal cut of the hierarchy by explicitly storing all of the nodes at a particular depth in a set. A better approach is to accept a function that reports whether or not a specified node is on the cut (an approach that works equally well for both horizontal and non-horizontal cuts).

4. The SimpleTug Method

We are now in a position to describe our *SimpleTug* method, which reformulates TugGraph in terms of the generalised multi-node unzipping algorithm we saw in §3. As mentioned in §2.3, the desired effect of tugging on a node is to separate out the parts of the base graph that are adjacent to the tugged node from their

The SimpleTug Method	
Inputs:	n, the node to tug $\mathcal{C}$, the non-horizontal cut being used to view the hierarchy
Auxiliary Definitions:	
$\mathcal{L}_{\mathcal{H}}(n) = \text{adjLeaves}_{\mathcal{H}}(n) = \{n' \in V_{\mathcal{H}} \setminus \mathcal{R}(n) : \exists n'' \in \mathcal{R}(n) \,.\, (\{n', n''\} \in E_{\mathcal{H}})\}$	
Main Definition:	
$\text{simpleTug}_{n;\mathcal{C}}(\mathcal{H}) = \text{unzip}_{\mathcal{L}_{\mathcal{H}}(n);\mathcal{C}}(\mathcal{H})$	

Table 6: Tugging on a node n in the context of a hierarchy that is being visualised using a non-horizontal cut $\mathcal{C}$ has the same effect as performing a single multi-node unzip of the leaves adjacent to n up to $\mathcal{C}$. This observation allows us to express our *SimpleTug* approach in closed form, which is helpful for proofs.

ancestors below the non-horizontal cut that is currently being used to view the hierarchy. Given this, steps 1–2 of the TugGraph algorithm can be seen as finding the parts of the base graph that need to be separated out (the 'adjacent leaves'), and steps 3–5 can be seen as effecting the separation, which can be achieved in a variety of different ways (of which the original TugGraph and *FastTug* are two examples). The key insight that we exploit to define *SimpleTug* is that this separation can be achieved efficiently and coherently using a single generalised unzip operation, as shown in Table 6 and Figure 4. This allows us to express the entire tugging process in closed form, as a generalised unzip of the adjacent leaves of the node n being tugged, up to the non-horizontal cut $\mathcal{C}$ being used to view the hierarchy.

This definition has a number of appealing properties. Firstly, like *FastTug* [19], it can be implemented in a way that avoids recreating parts of the intermediate hierarchy unnecessarily, but unlike that approach, it is modular, allowing us to preserve the hierarchy invariant throughout rather than breaking and then restoring it over the course of the operation, which is conceptually cleaner and also saves memory. Secondly, its expression in closed form makes it extremely convenient for proofs. Thirdly, it helps clarify what TugGraph is doing by expressing its behaviour in a straightforward way: tugging on a node means separating out (i.e. unzipping) the leaves surrounding it from their ancestors below the non-horizontal cut, and this definition clearly captures that intuition. Finally, this definition highlights the interesting theoretical connection between

the zipping algorithms for graph hierarchy editing described in [39, 13] and the TugGraph algorithm for graph hierarchy visualisation described in [4]: in particular, once we generalise unzipping to support non-horizontal cuts, it becomes clear that TugGraph can be regarded as an unzip of the adjacent leaf nodes.

4.1. Complexity Analysis

To analyse the computational complexity of *SimpleTug*, we follow [13] in denoting the cost of an arbitrary node-splitting operation as C_s, and the cost of an arbitrary sibling node merge operation as C_m. In practice, as mentioned in [13], these costs are themselves dependent on the way in which we choose to store higher-level edges in the hierarchy: we refer the reader to [13] for more details. We start by analysing the complexity of finding the leaves adjacent to the node being tugged, before analysing the complexity of generalised multi-node unzipping and the *SimpleTug* algorithm itself.

4.1.1. Complexity of Finding the Adjacent Leaves

Consider traversing down a hierarchy $\mathcal{H}$ from a specified node n to the set of leaves $N = \Psi^{-}_{\mathcal{H}|V_{\mathcal{H}}}(n)$ it subsumes. Traversing down from n to a single leaf $\ell \in N$ has complexity $O(\mathcal{D}(\ell) - \mathcal{D}(n))$. An upper bound for the cost of traversing down to all of the subsumed leaves is thus

$$O(|N|(\mathcal{D}_{\max}(N) - \mathcal{D}(n))).$$

Once the subsumed leaves have been found, the additional complexity of finding the leaves adjacent to them is then

$$O(|N||V_{\mathcal{H}}|),$$

since each leaf could in theory have degree $V_{\mathcal{H}}$. In real-world hierarchies, however, the maximum degree of a leaf is usually bounded by a fairly small constant, in which case this complexity reduces to $O(|N|)$. The overall complexity of finding $\mathcal{L}_{\mathcal{H}}(n)$, the adjacent leaves of n, is then dominated by the initial traversal

down the tree, and is hence

$$O(|N|(\mathcal{D}_{\max}(N) - \mathcal{D}(n))).$$

4.1.2. Complexity of Generalised Multi-Node Unzipping

Consider a generalised multi-node unzip that unzips the nodes in N up to a non-horizontal cut $\mathcal{C}$. In the worst case, all of the nodes in N will be at the same depth, namely $\mathcal{D}_{\max}(N)$, and each node in N will need to be unzipped individually up to depth $\mathcal{D}_{\min}(\mathcal{C}) + 1$. The worst-case complexity is thus

$$O(|N|(\mathcal{D}_{\max}(N) - \mathcal{D}_{\min}(\mathcal{C}))C_s).$$

4.1.3. Complexity of SimpleTug

Consider an application of *SimpleTug* that tugs on a node n in the context of a hierarchy that is being visualised using a non-horizontal cut $\mathcal{C}$. As per the definition, this involves first finding the adjacent leaves of n, and then performing a generalised unzip of the adjacent leaves up to the cut. In practice, the cost of finding the adjacent leaves is dominated by the cost of the unzip, and can be neglected. The complexity of the overall operation is thus

$$O(|\mathcal{L}_{\mathcal{H}}(n)|(\mathcal{D}_{\max}(\mathcal{L}_{\mathcal{H}}(n)) - \mathcal{D}_{\min}(\mathcal{C}))C_s).$$

Note that this result makes intuitive sense: the cost of the overall operation is based on (i) the number of leaves that are adjacent to n (the more there are, the more costly the operation), (ii) how high up the hierarchy the cut $\mathcal{C}$ is with respect to the adjacent leaves being unzipped (the higher it is, the more costly the operation), and (iii) the cost of an individual split.

4.2. Equivalence to FastTug

Having analysed *SimpleTug*'s complexity, we now seek to prove its equivalence to the original *FastTug* approach [19]. To do this, we first need to define *FastTug*

Formalising the FastTug Method of [19]

Step 4: Ripping Out

Inputs: N, the adjacent leaves of the node n that is being tugged
$\mathcal{C}$, the non-horizontal cut being used to view the hierarchy

Auxiliary Definitions:

$$\begin{aligned}
\Omega^d_{\mathcal{H}}(N, \mathcal{C}) &= \Psi^d_{\mathcal{H}}(N_{>d}) \cap \Psi^-_{\mathcal{H}}(\mathcal{C}) \\
\text{remnant}(n, N) &= (\mathcal{D}(n), \text{remnantRegion}(n, N)) \\
\text{ripAnc}_{\mathcal{H}}(d, N) &= \text{tugged}(d, N) \cup \{\text{remnant}(\psi^d_{\mathcal{H}}(N), N)\}
\end{aligned}$$

Main Definition:

$$\text{ripOut}_{N;\mathcal{C}}(\mathcal{H}) = \left(\text{ripOut}_{N;\mathcal{C}}(V^d_{\mathcal{H}}) : V^d_{\mathcal{H}} \in \mathcal{H}\right)$$

$$\text{ripOut}_{N;\mathcal{C}}(V^d_{\mathcal{H}}) = \begin{cases} (V^d_{\mathcal{H}} \setminus \Omega^d_{\mathcal{H}}(N, \mathcal{C})) \cup \\ \quad \bigcup \left\{\text{ripAnc}_{\mathcal{H}}(d, \Psi^-_{\mathcal{H}|N}(n)) : n \in \Omega^d_{\mathcal{H}}(N, \mathcal{C})\right\} & \text{if } d \in (\mathcal{D}_{\min}(\mathcal{C}), \mathcal{D}_{\max}(N)) \\ V^d_{\mathcal{H}} & \text{otherwise} \end{cases}$$

Step 5: Fixing Up

Inputs: Q, a priority queue of nodes that need fixing up, sorted in non-increasing depth order

Auxiliary Definitions:

$$\text{fixNode}(n) = \{(\mathcal{D}(n), R') : R' \in \text{ccs}(\mathcal{R}(n))\}$$

Main Definition:

$$\text{fixUp}_Q(\mathcal{H}) = \left(\text{fixUp}_Q(V^d_{\mathcal{H}}) : V^d_{\mathcal{H}} \in \mathcal{H}\right)$$

$$\text{fixUp}_Q(V^d_{\mathcal{H}}) = \begin{cases} (V^d_{\mathcal{H}} \setminus Q_{=d}) \cup \bigcup\{\text{fixNode}(r) : r \in Q_{=d}\} & \text{if } d \in (\mathcal{D}_{\min}(\mathcal{C}), \mathcal{D}_{\max}(N)) \\ V^d_{\mathcal{H}} & \text{otherwise} \end{cases}$$

The Overall Method

Inputs: n, the node to tug
$\mathcal{C}$, the non-horizontal cut being used to view the hierarchy

Auxiliary Definitions:

$$Q_{\mathcal{H}}(N, \mathcal{C}) = \text{sort}(\{\text{remnant}(n, \Psi^-_{\mathcal{H}|N}(n)) : d \in (\mathcal{D}_{\min}(\mathcal{C}), \mathcal{D}_{\max}(N)), n \in \Omega^d_{\mathcal{H}}(N, \mathcal{C})\})$$

Main Definition:

$$\text{fastTug}_{n;\mathcal{C}}(\mathcal{H}) = \text{fixUp}_{Q_{\mathcal{H}}(\mathcal{L}_{\mathcal{H}}(n), \mathcal{C})}(\text{ripOut}_{\mathcal{L}_{\mathcal{H}}(n);\mathcal{C}}(\mathcal{H}))$$

Table 7: A mathematical formulation of the *FastTug* approach to TugGraph from [19], which allows a node n to be tugged in the context of a hierarchy that is being visualised using a non-horizontal cut $\mathcal{C}$. The method first finds the adjacent leaves $\mathcal{L}_{\mathcal{H}}(n)$ in the usual manner (see §2.3 and Figure 1). It then moves them directly into nodes just below the cut ('ripping out'), before splitting any nodes that have become disconnected as necessary to restore the hierarchy invariant ('fixing up').

formally (see Table 7), since the original paper did not include a mathematical formulation of the algorithm. We split this into three parts (referring back to the initial description of *FastTug* in §2.4): (i) the movement of the adjacent leaves directly into nodes just below the cut (which we call 'ripping out'), (ii) the splitting of nodes as necessary to restore the hierarchy invariant (which we call 'fixing up'), and (iii) the overall method, which is responsible both for finding the adjacent leaves, and for combining the ripping out and fixing up processes appropriately.

The essence of the proof is to show that the composition of the 'ripping out' and 'fixing up' operations is equivalent to unzipping. To understand intuitively why this must be the case, observe that at each depth d, 'ripping out' splits each node $n \in \Omega^d_{\mathcal{H}}(N, \mathcal{C})$ around its descendants in N and replaces it with two types of node: (a) 'tugged' nodes (as also used in ancestor splitting in §2.5), which are guaranteed to be connected, and (b) a 'remnant' node, which is only connected if what is left of n after removing the tugged nodes forms a single connected component. This contrasts with unzipping, which computes the connected components of what is left of n, and creates a remnant node for each such component. The 'fixing up' process bridges the gap between the two, computing the connected components of each remnant node produced by 'ripping out' and replacing it with a node for each such component; however, unlike with unzipping, this happens in a second pass over the hierarchy, as shown in Table 7.

To show the equivalence formally, we start by proving that fixing up the remnant node produced by the ripping out process results in the same remnant nodes that would have been produced by unzipping:

Lemma 1. *fixNode*$(\mathit{remnant}(n, N)) = \mathit{remnants}(n, N)$.

Proof.

$$\begin{aligned}
&\text{fixNode}(\text{remnant}(n, N)) \\
&= \{(\mathcal{D}(\text{remnant}(n, N)), R') : R' \in \text{ccs}(\mathcal{R}(\text{remnant}(n, N)))\} \\
&= \{(\mathcal{D}(n), R') : R' \in \text{ccs}(\text{remnantRegion}(n, N))\} \\
&= \text{remnants}(n, N)
\end{aligned}$$

□

Next, we show that the composition of the two processes is equivalent to unzipping for each depth d in the hierarchy:

Lemma 2. $\mathit{fixUp}_{Q_{\mathcal{H}}(N,\mathcal{C})}(\mathit{ripOut}_{N;\mathcal{C}}(V^d_{\mathcal{H}})) = \mathit{unzip}_{N;\mathcal{C}}(V^d_{\mathcal{H}})$.

Proof. The proof for $d \notin (\mathcal{D}_{\min}(\mathcal{C}), \mathcal{D}_{\max}(N))$ is trivial. For the more interesting case, we first observe that:

$$\begin{aligned}
&\text{ripOut}_{N;\mathcal{C}}(V^d_{\mathcal{H}}) \\
&= (V^d_{\mathcal{H}} \setminus \Omega^d_{\mathcal{H}}(N, \mathcal{C})) \cup \bigcup \left\{ \text{ripAnc}_{\mathcal{H}}(d, \Psi^-_{\mathcal{H}|N}(n)) : n \in \Omega^d_{\mathcal{H}}(N, \mathcal{C}) \right\} \\
&= (V^d_{\mathcal{H}} \setminus \Omega^d_{\mathcal{H}}(N, \mathcal{C})) \cup \bigcup \left\{ \text{tugged}(d, \Psi^-_{\mathcal{H}|N}(n)) : n \in \Omega^d_{\mathcal{H}}(N, \mathcal{C}) \right\} \\
&\quad \cup \bigcup \left\{ \text{remnant}(n, \Psi^-_{\mathcal{H}|N}(n)) : n \in \Omega^d_{\mathcal{H}}(N, \mathcal{C}) \right\}
\end{aligned}$$

Now the definition of *fixUp* (see Table 7) works by fixing up the nodes in a priority queue Q at each depth d. In this case, Q is $Q_{\mathcal{H}}(N, \mathcal{C})$, and we have:

$$(Q_{\mathcal{H}}(N, \mathcal{C}))_{=d} = \left\{ \text{remnant}(n, \Psi^-_{\mathcal{H}|N}(n)) : n \in \Omega^d_{\mathcal{H}}(N, \mathcal{C}) \right\}$$

But then:

$$\begin{aligned}
&\text{fixUp}_{Q_{\mathcal{H}}(N,\mathcal{C})}(\text{ripOut}_{N;\mathcal{C}}(V^d_{\mathcal{H}})) \\
&= (V^d_{\mathcal{H}} \setminus \Omega^d_{\mathcal{H}}(N, \mathcal{C})) \cup \bigcup \left\{ \text{tugged}(d, \Psi^-_{\mathcal{H}|N}(n)) : n \in \Omega^d_{\mathcal{H}}(N, \mathcal{C}) \right\} \\
&\quad \cup \bigcup \left\{ \text{fixNode}(\text{remnant}(n, \Psi^-_{\mathcal{H}|N}(n))) : n \in \Omega^d_{\mathcal{H}}(N, \mathcal{C}) \right\} \\
&= (V^d_{\mathcal{H}} \setminus \Omega^d_{\mathcal{H}}(N, \mathcal{C})) \cup \bigcup \left\{ \text{tugged}(d, \Psi^-_{\mathcal{H}|N}(n)) : n \in \Omega^d_{\mathcal{H}}(N, \mathcal{C}) \right\} \\
&\quad \cup \bigcup \left\{ \text{remnants}(n, \Psi^-_{\mathcal{H}|N}(n)) : n \in \Omega^d_{\mathcal{H}}(N, \mathcal{C}) \right\} \\
&= (V^d_{\mathcal{H}} \setminus \Omega^d_{\mathcal{H}}(N, \mathcal{C})) \cup \bigcup \left\{ \text{splitAnc}_{\mathcal{H}}(d, \Psi^-_{\mathcal{H}|N}(n)) : n \in \Omega^d_{\mathcal{H}}(N, \mathcal{C}) \right\} \\
&= \text{unzip}_{N;\mathcal{C}}(V^d_{\mathcal{H}})
\end{aligned}$$

□

We can lift this proof to the full hierarchy as follows:

Parallel Tugging	
Inputs:	N, the nodes to tug $\mathcal{C}$, the non-horizontal cut being used to view the hierarchy
Auxiliary Definitions:	
$\mathcal{L}_{\mathcal{H}}(N) = \text{adjLeaves}_{\mathcal{H}}(N) = \{n' \in V_{\mathcal{H}} \backslash \mathcal{R}(N) : \exists n'' \in \mathcal{R}(N).(\{n', n''\} \in E_{\mathcal{H}})\}$	
Main Definitions:	
$\text{fastTug}_{N;\mathcal{C}}(\mathcal{H}) = \text{fixUp}_{Q_{\mathcal{H}}(\mathcal{L}_{\mathcal{H}}(N),\mathcal{C})}(\text{ripOut}_{\mathcal{L}_{\mathcal{H}}(N);\mathcal{C}}(\mathcal{H}))$	
$\text{simpleTug}_{N;\mathcal{C}}(\mathcal{H}) = \text{unzip}_{\mathcal{L}_{\mathcal{H}}(N);\mathcal{C}}(\mathcal{H})$	

Table 8: Both the *FastTug* and *SimpleTug* algorithms can be naturally parallelised by generalising the function $\mathcal{L}_{\mathcal{H}}$ that computes the adjacent leaves of the node being tugged to accept a set of nodes rather than just a single node. In the case of *FastTug*, parallelism is only possible for concurrent tugs that start at the same time, whereas *SimpleTug* does not have this restriction: see §5 for more details.

Theorem 3. *$fixUp_{Q_{\mathcal{H}}(N,\mathcal{C})}(ripOut_{N;\mathcal{C}}(\mathcal{H})) = unzip_{N;\mathcal{C}}(\mathcal{H})$.*

Proof.

$$
\begin{aligned}
&\text{fixUp}_{Q_{\mathcal{H}}(N,\mathcal{C})}(\text{ripOut}_{N;\mathcal{C}}(\mathcal{H})) \\
&= \text{fixUp}_{Q_{\mathcal{H}}(N,\mathcal{C})}\left((\text{ripOut}_{N;\mathcal{C}}(V^d_{\mathcal{H}}) : V^d_{\mathcal{H}} \in \mathcal{H})\right) \\
&= \left(\text{fixUp}_{Q_{\mathcal{H}}(N,\mathcal{C})}(\text{ripOut}_{N;\mathcal{C}}(V^d_{\mathcal{H}})) : V^d_{\mathcal{H}} \in \mathcal{H}\right) \\
&= \left(\text{unzip}_{N;\mathcal{C}}(V^d_{\mathcal{H}}) : V^d_{\mathcal{H}} \in \mathcal{H}\right) \\
&= \text{unzip}_{N;\mathcal{C}}(\mathcal{H})
\end{aligned}
$$

□

Finally, we can use Theorem 3 to show that *FastTug* (which involves fixing up the results of ripping out the adjacent leaves $\mathcal{L}_{\mathcal{H}}(n)$ of n) and *SimpleTug* (which involves unzipping the same adjacent leaves) achieve the same results, which was what we needed to prove:

Corollary 4. *$fastTug_{n;\mathcal{C}}(\mathcal{H}) = simpleTug_{n;\mathcal{C}}(\mathcal{H})$.*

Proof. Trivial (substitute $\mathcal{L}_{\mathcal{H}}(n)$ for N in Theorem 3). □

5. Parallel Tugging

For many applications, using either *FastTug* or *SimpleTug* to tug on a single node at a time may be sufficient, but in some cases, it may be desirable to tug on multiple nodes at the same time. As shown in Table 8, the formulations we give for both algorithms in §4 generalise naturally to this use case: it suffices

to simply generalise the function that computes the adjacent leaves to accept a set of nodes rather than just a single node. It might initially seem surprising that *FastTug* can be parallelised in this way, given the way in which it initially breaks the hierarchy invariant and only restores it over the course of the entire operation, and indeed, it would not be possible to interleave concurrent tugs that start at *different* times for this reason. However, it *is* possible to parallelise concurrent tugs that start at the *same* time, since in that case the invariant is only broken once (at the start, by all the tugs that are being jointly performed) and then restored over the course of the operation as normal. It is this that we exploit in formulating a parallel version of *FastTug* in this way.

As we show in §6, tugging large numbers of nodes in parallel (with either algorithm) has significant time advantages over tugging them one after another. Interestingly, however, parallelising the two algorithms exacerbates the space advantage that *SimpleTug* already has over *FastTug*, owing to the latter's need to store the nodes that need to be fixed up as the operation proceeds (when large numbers of nodes are being tugged, storing the nodes to be fixed up can become somewhat costly).

6. Experiments

In §4.2, we formally proved that our *SimpleTug* method produces the same results as the earlier approaches to TugGraph, but that on its own tells us little about its effectiveness in practice: in particular, the time and space requirements of different implementations of the TugGraph concept can vary significantly, and this property is only weakly captured by the mathematical definitions given thus far. For this reason, we implemented serial and parallel variants of both *SimpleTug* and the original *FastTug* algorithm [19] in our *millipede* hierarchical image segmentation system[6] [39, 13], and performed some quantitative experiments on them to evaluate the performance of our approach.

[6]Our open-source *millipede* system was originally developed to help clinicians semi-automatically segment organs and other features in abdominal CT scans [39]; it was later

We used *millipede* to construct several graph hierarchies of different heights (10, 100 and 1000) for each of the 500 images in the Berkeley Segmentation Data Set (BSDS500) [41],[7] and then determined the average times it took each algorithm to tug on varying numbers of randomly-chosen nodes from the hierarchies, both in serial and in parallel. In other words, for each graph hierarchy, we enumerated all of the nodes, chose random subsets of the nodes of different sizes, and then timed the same tugs with each algorithm implementation. We also determined the amounts of additional memory needed by the serial and parallel implementations of *FastTug*, in comparison to their *SimpleTug* equivalents, in each case. Finally, we averaged all the numbers (that is, both the time and space numbers, for all implementations) over the entire dataset.

To actually build the hierarchies, we used the deep hierarchy construction approach described in [13]: this takes a desired height for the hierarchy, and calculates the ideal rate at which to merge nodes in each hierarchy layer so as to achieve convergence to a single node at the chosen height. This approach has the advantage of producing interesting hierarchies, whilst being much simpler to implement than more sophisticated approaches like the waterfall algorithm [42, 40] or MCG [43], and giving us full control over the hierarchy's height, which is helpful for evaluation purposes.

6.1. Timings

Our timing results are shown in Table 9. In all cases (for both the serial and parallel versions of both *FastTug* and *SimpleTug*), the time taken increased significantly with either increasing numbers of nodes being tugged, or increasing hierarchy height (indeed, tugging 1000 randomly-chosen nodes in serial in each of 500 hierarchies of height 1000 took more than 10 days on an Intel i7-7820X

extended to support hierarchical segmentation of any 2D or 3D image [40]. The implementation is in C++, and makes use of the ITK library for low-level image processing. User interaction is via a wxWidgets-based GUI, a detailed description of which can be found in [13]. Source code can be found online at https://github.com/sgolodetz/millipede/tree/millipede_v2.

[7]The BSDS500 is an appropriate dataset on which to perform these comparisons because it contains a wide variety of natural images that we would expect to yield an interesting range of different hierarchical structures.

FastTug Timings (in milliseconds)

		Serial				Parallel		
Nodes Tugged ⟶		*1*	*10*	*100*	*1000*	*10*	*100*	*1000*
Height	*10*	9	79	1026	10891	51	275	1282
	100	32	377	6853	167818	257	1909	11378
	1000	250	3281	64068	1758982	2111	18447	148875

SimpleTug Timings (in milliseconds)

		Serial				Parallel		
Nodes Tugged ⟶		*1*	*10*	*100*	*1000*	*10*	*100*	*1000*
Height	*10*	7	60	800	8294	40	208	932
	100	27	290	5084	112458	189	1430	8421
	1000	206	2446	47008	1201134	1530	13451	91954

Table 9: The average times (in milliseconds) taken by the serial and parallel versions of *FastTug* and *SimpleTug* to tug on different numbers of randomly-chosen nodes from deep hierarchies [13] of different heights, averaged over the 500 images of the Berkeley Segmentation Data Set (BSDS500) [41]. Note that *SimpleTug* is consistently faster than *FastTug* in all cases, by around 25% on average, whilst producing exactly the same results (see §4.2).

CPU). Two important observations can be made, however. Firstly, both the serial and parallel versions of *SimpleTug* are significantly faster than their *FastTug* counterparts, by around 25% on average. This is due to the fact that in comparison to *FastTug*, our *SimpleTug* approach removes the need to construct a priority queue of disconnected nodes and restore their connectivity in a second pass, reducing the overall amount of work involved in tugging a node. Secondly, as one might expect, parallel tugging was dramatically faster than serial tugging when tugging large numbers of nodes, for both *FastTug* and *SimpleTug*, although *SimpleTug* always maintains a significant edge in practice.

6.2. Memory Usage

The amounts of additional memory needed by the serial and parallel implementations of *FastTug* during these experiments, in comparison to their *SimpleTug* equivalents, are shown in Table 10. The additional memory is needed to maintain a priority queue of disconnected nodes that need to be fixed up. When tugging in serial, or tugging a small number of nodes, relatively little additional memory is needed. However, when tugging a large number of nodes with the parallel implementation of *FastTug*, significant additional memory (tens of megabytes) is required, since then the number of nodes that need to be fixed

FastTug Additional Memory Usage								
		Serial				**Parallel**		
Nodes Tugged ⟶		*1*	*10*	*100*	*1000*	*10*	*100*	*1000*
Height	*10*	754B	813B	939B	774B	8KB	86KB	557KB
	100	10KB	10KB	12KB	11KB	96KB	1.1MB	7.4MB
	1000	91KB	103KB	126KB	110KB	1002KB	11.5MB	74.8MB

Table 10: The additional memory used by the serial and parallel versions of *FastTug* in comparison to the corresponding versions of *SimpleTug*, whilst tugging on different numbers of randomly-chosen nodes from deep hierarchies [13] of different heights, averaged over the 500 images of the Berkeley Segmentation Data Set (BSDS500) [41]. Both versions of *FastTug* use more memory than their *SimpleTug* equivalents. Note that whilst the additional memory needed in the serial case is quite minor, neither serial *FastTug* nor serial *SimpleTug* scales well to tugging many nodes on very deep hierarchies (see Table 9). The parallel algorithms scale better, but parallel *FastTug* uses much more memory than parallel *SimpleTug*.

up can be quite large. By contrast, neither the serial nor the parallel versions of *SimpleTug* have a need to maintain a priority queue of disconnected nodes, allowing them to save memory, and enabling the speed gains of parallel tugging to be leveraged without a significant memory cost.

7. Discussion

The way in which *SimpleTug* reformulates the original TugGraph algorithm [18] as a multi-node unzip of the leaves adjacent to the tugged node reveals an interesting theoretical connection between the zipping algorithms of [13], which were designed to be used as reusable mid-level building blocks for graph hierarchy editing, and TugGraph, which was originally designed as a special-purpose algorithm for visualising the surroundings of a particular graph hierarchy node. In [13], we described two potential applications of our zipping algorithms: non-sibling node merging, which separates spatially adjacent nodes out from their ancestors in order to merge the resulting chains of nodes, and parent switching, which separates out a single node in order to transfer it to a new parent. In this paper, we have shown that TugGraph can be seen as a third, distinct application of our zipping algorithms, one that separates out the adjacent leaves of a particular node to allow them to be viewed by the user. Moreover, we have shown that viewing TugGraph in this light allows us to both understand and implement it more effectively, since by reformulating it in terms of our in-

herently decomposable zipping algorithms, we can allow TugGraph itself to be decomposed.

It is interesting to consider whether further applications of our zipping algorithms exist. Abstractly, the existing applications we have described can effectively be seen as (i) separating out nodes to combine them to make a new group, (ii) separating out a node to move it from one group to another, and (iii) separating out nodes to see them better. Other reasons for separating out nodes certainly exist: for example, we might want to separate out one or more nodes to deliberately disconnect their ancestors. However, in practice, the existing applications we have described already cover a significant part of the relevant problem space.

Despite this, we believe there are at least a couple of potentially interesting avenues for further work in this area. As mentioned in the introduction, one advantage of modularising TugGraph is that it makes it possible to allow several different users to perform concurrent, interleaved tugs on nodes in the hierarchy, e.g. in a cloud visualisation context. As such, one possible avenue might be to extend the existing TugGraph system to support this paradigm. A second interesting avenue might be to study the uses of the zipping algorithms for editing graph hierarchies in other contexts, such as hierarchical pathfinding [44] for games and transport applications. Such hierarchies are often constructed manually, at some expense [45], and it would be interesting to investigate whether the zipping algorithms can be used to facilitate more automated refinement of the hierarchies to reduce this cost.

8. Conclusion

The TugGraph algorithm is an important and effective technique for visualising the local structure around a node in a graph hierarchy, but both existing ways of implementing it have drawbacks: the original approach [18], whilst easy to understand and implement, is comparatively slow and memory-hungry due to the hierarchy-copying it involves, and the same authors' later (optimised) approach

[19], whilst much faster and using less memory, is in practice quite complicated, owing to the difficulties inherent in modifying a graph hierarchy efficiently whilst maintaining its path-preserving structure (the hierarchy invariant).

Recently, however, such difficulties have been overcome (in a different context) through the introduction of so-called 'zipping' algorithms for graph hierarchies [13]. These function as multi-layer split and merge algorithms with low input requirements, and are intended to be used as intermediate-level building blocks to simplify the implementation of existing hierarchy editing algorithms.

In this paper, we have shown how these algorithms can be generalised to achieve an implementation of TugGraph that is fast, memory-efficient, and easy to understand and implement. Moreover, we have shown that by reformulating the algorithm in this way, it is possible to gain additional insight into its effects on a hierarchy: ultimately, the aim when tugging a node is to separate out the leaf nodes adjacent to that node from their containing nodes on the cut. This can be expressed both straightforwardly and naturally using generalised multi-node unzipping.

Acknowledgements

This paper is dedicated to the memory of Stephen Cameron, who sadly passed away whilst it was under review. Stuart Golodetz and Anurag Arnab are grateful to Philip Torr for letting them undertake this work during their time in his group. Stuart Golodetz would also like to reiterate his thanks to EPSRC for funding his Doctoral Training Award (DTA).

References

[1] I. Herman, G. Melançon, M. S. Marshall, Graph Visualization and Navigation in Information Visualization: A Survey, IEEE Transactions on Visualization and Computer Graphics 6 (1) (2000) 24–43. 2

[2] C. Wang, J. Tao, Graphs in Scientific Visualization: A Survey, Computer Graphics Forum 36 (1) (2017) 263–287. 2

[3] Google, TensorBoard: Graph Visualization, available online (as of 29th October 2019) at https://www.tensorflow.org/tensorboard/r1/graphs. 2

[4] D. Archambault, T. Munzner, D. Auber, GrouseFlocks: Steerable Exploration of Graph Hierarchy Space, IEEE Transactions on Visualization and Computer Graphics 14 (4) (2008) 900–913. 3, 8, 10, 19

[5] N. Elmqvist, T.-N. Do, H. Goodell, N. Henry, ZAME: Interactive Large-Scale Graph Visualization, in: IEEE Pacific Visualization Symposium, 2008, pp. 215–222. 3

[6] B. Schneiderman, Tree visualization with Tree-maps: A 2-d space-filling approach, ACM Transactions on Graphics 11 (1) (1992) 92–99. 3

[7] J. Yang, M. O. Ward, E. A. Rundensteiner, InterRing: An Interactive Tool for Visually Navigating and Manipulating Hierarchical Structures, in: IEEE Symposium on Information Visualization, 2002, pp. 77–84. 3

[8] N. Elmqvist, J.-D. Fekete, Hierarchical Aggregation for Information Visualization: Overview, Techniques and Design Guidelines, IEEE Transactions on Visualization and Computer Graphics 16 (3) (2010) 439–454. 3, 5

[9] S. Zhao, M. J. McGuffin, M. H. Chignell, Elastic Hierarchies: Combining Treemaps and Node-Link Diagrams, in: IEEE Symposium on Information Visualization, 2005, pp. 57–64. 3

[10] N. Henry, J.-D. Fekete, MatrixExplorer: a Dual-Representation System to Explore Social Networks, IEEE Transactions on Visualization and Computer Graphics 12 (5) (2006) 677–684. 3

[11] N. Henry, J.-D. Fekete, M. McGuffin, NodeTrix: a Hybrid Visualization of Social Networks, IEEE Transactions on Visualization and Computer Graphics 13 (6) (2007) 1302–1309. 3

[12] L. Angori, W. Didimo, F. Montecchiani, D. Pagliuca, A. Tappini, ChordLink: A New Hybrid Visualization Model, in: International Symposium on Graph Drawing and Network Visualization, 2019, pp. 276–290. 3

[13] S. Golodetz, I. Voiculescu, S. Cameron, Simpler Editing of Graph-Based Segmentation Hierarchies using Zipping Algorithms, Pattern Recognition 70 (2017) 44–59. 3, 4, 5, 6, 7, 12, 13, 14, 15, 16, 19, 25, 26, 27, 28, 30

[14] R. Ying, J. You, C. Morris, X. Ren, W. L. Hamilton, J. Leskovec, Hierarchical Graph Representation Learning with Differentiable Pooling, in: Advances in Neural Information Processing Systems, 2018, pp. 4800–4810. 3

[15] J. F. Randrianasoa, C. Kurtz, E. Desjardin, N. Passat, Binary partition tree construction from multiple features for image segmentation, Pattern Recognition 84 (2018) 237–250. 3

[16] G. Tochon, M. D. Mura, M. A. Veganzones, T. Géraud, J. Chanussot, Braids of partitions for the hierarchical representation and segmentation of multimodal images, Pattern Recognition 95 (2019) 162–172. 3

[17] S. Golodetz, I. Voiculescu, S. Cameron, Automatic Spine Identification in Abdominal CT Slices using Image Partition Forests, in: International Symposium on Image and Signal Processing and Analysis, 2009, pp. 117–122. 3

[18] D. Archambault, T. Munzner, D. Auber, TugGraph: Path-Preserving Hierarchies for Browsing Proximity and Paths in Graphs, in: IEEE Pacific Visualization Symposium, 2009, pp. 113–121. 3, 5, 6, 8, 9, 10, 28, 29

[19] D. Archambault, T. Munzner, D. Auber, Tugging Graphs Faster: Efficiently Modifying Path-Preserving Hierarchies for Browsing Paths, IEEE Transactions on Visualization and Computer Graphics 17 (3) (2011) 276–289. 3, 6, 10, 11, 12, 14, 18, 20, 21, 25, 30

[20] G. Booch, Object-Oriented Analysis and Design, 2nd Edition, Addison-Wesley, 1994, pp. 14–16. 4

[21] T. V. Landesberger, A. Kuijper, T. Schreck, J. Kohlhammer, Visual Analysis of Large Graphs: State-of-the-Art and Future Research Challenges, Computer Graphics Forum 30 (6) (2012) 1719–1749. 5

[22] R. Pienta, J. Abello, M. Kahng, D. H. Chau, Scalable Graph Exploration

and Visualization: Sensemaking Challenges and Opportunities, in: International Conference on Big Data and Smart Computing (BigComp), 2015, pp. 271–278. 5

[23] C. Vehlow, F. Beck, D. Weiskopf, Visualizing Group Structures in Graphs: A Survey, Computer Graphics Forum 36 (6) (2016) 201–225. 5

[24] B. Klava, N. S. T. Hirata, Watershed segmentation: Switching back and forth between markers and hierarchies, in: International Symposium on Mathematical Morphology, 2007, pp. 29–30. 7

[25] J. Abello, F. van Ham, N. Krishnan, ASK-GraphView: A Large Scale Graph Visualisation System, IEEE Transactions on Visualization and Computer Graphics 12 (5) (2006) 669–676. 7

[26] J. S. Yi, Y. ah Kang, J. T. Stasko, J. A. Jacko, Toward a Deeper Understanding of the Role of Interaction in Information Visualization, IEEE Transactions on Visualization and Computer Graphics 13 (6) (2007) 1224–1231. 7

[27] C. Tominski, J. Abello, H. Schumann, CGV – An interactive graph visualization system, Computers & Graphics 33 (6) (2009) 660–678. 7

[28] S. Gladisch, H. Schumann, C. Tominski, Navigation Recommendations for Exploring Hierarchical Graphs, in: International Symposium on Visual Computing, 2013, pp. 36–47. 8

[29] J. Zhang, A. Malik, B. Ahlbrand, N. Elmqvist, R. Maciejewski, D. S. Ebert, TopoGroups: Context-Preserving Visual Illustration of Multi-Scale Spatial Aggregates, in: ACM CHI Conference on Human Factors in Computing Systems, 2017, pp. 2940–2951. 8

[30] R. AlTarawneh, J. Schultz, S. R. Humayoun, CluE: An Algorithm for Expanding Clustered Graphs, in: IEEE Pacific Visualization Symposium, 2014, pp. 233–237. 8

[31] D. Auber, F. Jourdan, Interactive Refinement of Multi-scale Network Clusterings, in: International Conference on Information Visualisation, 2005, pp. 703–709. 8

[32] P. Eades, M. L. Huang, Navigating Clustered Graphs using Force-Directed Methods, Journal of Graph Algorithms and Applications 4 (3) (2000) 157–181. 8

[33] D. H. Fisher, Knowledge Acquisition Via Incremental Conceptual Clustering, Machine Learning 2 (2) (1987) 139–172. 8

[34] S. Golodetz, I. Voiculescu, S. Cameron, Region Analysis of Abdominal CT Scans using Image Partition Forests, in: International Conference on Soft Computing as Transdisciplinary Science and Technology, 2008, pp. 432–7. 8

[35] P. F. M. Nacken, Image Segmentation by Connectivity Preserving Relinking in Hierarchical Graph Structures, Pattern Recognition 28 (6) (1995) 907–920. 8

[36] J. Cousty, L. Najman, Morphological Floodings and Optimal Cuts in Hierarchies, in: IEEE International Conference on Image Processing, 2014, pp. 4462–4466. 8

[37] M. Maire, S. X. Yu, P. Perona, Hierarchical Scene Annotation, in: British Machine Vision Conference, 2013. 8

[38] G. Richer, J. Sansen, F. Lalanne, D. Auber, R. Bourqui, HiePaCo: Scalable Hierarchical Exploration in Abstract Parallel Coordinates Under Budget Constraints, Big Data Research 17 (2019) 1–17. 8

[39] S. Golodetz, Zipping and Unzipping: The Use of Image Partition Forests in the Analysis of Abdominal CT Scans, Ph.D. thesis, University of Oxford (2011). 19, 25

[40] S. Golodetz, C. Nicholls, I. Voiculescu, S. Cameron, Two Tree-Based Methods for the Waterfall, Pattern Recognition 47 (10) (2014) 3276–3292. 26

[41] P. Arbeláez, M. Maire, C. Fowlkes, J. Malik, Contour Detection and Hierarchical Image Segmentation, IEEE Transactions on Pattern Analysis and Machine Intelligence 33 (5) (2011) 898–916. 26, 27, 28

[42] S. Beucher, Watershed, Hierarchical Segmentation and Waterfall Algo-

rithm, in: International Symposium on Mathematical Morphology, 1994, pp. 69–76. 26

[43] P. Arbeláez, J. Pont-Tuset, J. T. Barron, F. Marques, J. Malik, Multiscale Combinatorial Grouping, in: IEEE Conference on Computer Vision and Pattern Recognition, 2014, pp. 328–335. 26

[44] K. Kim, S. Yoo, S. K. Cha, A Partitioning Scheme for Hierarchical Path Finding Robust to Link Cost Update, in: Proceedings of the 5th World Congress on Intelligent Transport Systems (CD-ROM), 1998. 29

[45] M. Dickheiser, Inexpensive Precomputed Pathfinding Using a Navigation Set Hierarchy, in: S. Rabin (Ed.), AI Game Programming Wisdom 2, Charles River Media, 2004, pp. 103–113. 29